TD MAPLE n°1 PROGRAMMATION

I - Avant de commencer

Quelques rappels avant de travailler. Commencez chaque nouvel exercice par un

```
restart;
```

pour « désaffecter » toutes les variables éventuellement utilisées dans un calcul précédent.
Si vous avez lancé un calcul qui a l'air de ne pas vouloir se terminer, cliquez sur l'icône STOP qui n'est active que lorsqu'un calcul est en cours.
Chacune de vos entrées débute après un prompt >, se termine par un ; [a] ou un : [b] puis appuyez sur la touche Entrée. Si vous voulez passer à la ligne sans valider la ligne précédente, tapez Shift ⇧ + Entrée. Pour revenir plus haut, utilisez la souris ou les flèches de déplacement.

II - Listes et séquences

a. Listes

Les listes sont un type de variable particulier que peut manipuler MAPLE. Une liste se présente sous la forme d'une suite d'objets séparés par des virgules entre crochets :

```
> L:=[1,2,a,g,maman,ln(5),3/2,x->3*x+2,[7,9]]
```

Cette suite est ORDONNÉE. Ses éléments sont donc numérotés. On peut en extraire un élément particulier :

```
> L[5]
```

ou bien tous les éléments de la liste :

```
op(L)
```

ou bien une sous-liste :

```
> L[3..6];
```

On peut connaître la taille d'une liste :

```
> nops(L);
```

On peut ajouter un élément à une liste :

```
> L:=[op(L),nouveau]
```

On peut *concaténer* deux listes :

```
> L1:=[1,2,3,4];
> L2:=[a,b,c];
> L:=[op(L1),op(L2)]
```

La liste vide se note :

```
> V:=[];
> nops(V);
```

On peut effectuer des opérations sur une liste car c'est un type de variable comme un autre :

```
> S:=L1+[op(L2),d];
> P:=3*L1;
```

a. si vous voulez que le résultat soit affiché.
b. si vous voulez que MAPLE exécute sans afficher.

b. Les séquences

Elles ressemblent aux listes mais sans les crochets. La différence majeure est pourtant qu'on ne peut pas créer des séquences de séquences alors qu'on peut créer des listes de listes.
Plus accessoirement, on peut modifier un élément d'une liste à l'aide d'une affectation mais ce n'est pas possible pour une séquence :

```
> S:=a,b,c,d,e,f;
> S[3];
> S[3]:=4;
> L:=[S];
> L[3]:=4;
> L;
```

En général, elles servent à fabriquer des listes...

```
> S1:=10,20,30,40,50,60;
> S2:=seq(10*k,k=3..10);
> S3:=10*k $ k=2..9;
> S4:= m $ 5;
```

III - Tests

a. Expressions booléennes

Une algèbre de Boole est un ensemble contenant deux éléments et muni de deux lois de composition interne ET et OU.
En informatique, un booléen est un type de variable à deux états : « true » et « false » sur MAPLE.
Les opérateurs logiques sont and et or. Il y a aussi not qui donne la négation.
Par exemple :

```
> (3>2) and (2=5);
> (3>2) or (2=5);
> (3>2) and not (2=5);
```

On peut tester l'ordre et l'égalité avec des opérateurs relationnels : <, <=, >, >=, =, <>.
Quand une expression ne contient pas and, or ou not, il faut utiliser evalb ou is :

```
> evalb(3>2);
```

type de nombre

On ne peut comparer que des nombres de type numeric (i.e. integer, float et fraction) :

```
> evalb(3>2.);
> evalb(3>7/2);
> evalb(sqrt(2)>1);
> evalb(sqrt(2.)>1);
```

b. if...then...else

On utilise la valeur d'un test pour exécuter une instruction conditionnelle dont la syntaxe est :

if condition **then** instruction1 **else** instruction2 **fi** ;

Par exemple :

```
> if (3>2) then print('bonjour') else print('salut') fi;
> if (3<2) then print('bonjour') else print('salut') fi;
```

On peut écrire des instructions conditionnelles en cascade :

if condition1 **then** instruction1 **elif** condition2 **then** instruction2 **else** instruction3 **fi** ;

Par exemple :

```
> a:=1; b:=1; c:=1;

> if (b^2-4*a*c>0)
    then print(‘2 solutions réelles‘)
  elif (b^2-4*a*c=0)
    then print(‘1 solution réelle‘)
    else print(‘2 solutions complexes conjuguées‘)
  fi;
```

IV - Procédures

Une procédure est une fonction à la syntaxe précise dépendant de la donnée d'une série d'arguments et d'une série d'instructions.
La syntaxe générale est :

```
> nom:=proc(argument1::type1,argument2::type2,.....)
  local var1, var2,... ;
  instructions
  end:
```

Les principaux types sont `float`, `integer`, `negint`, `nonnegint`, `fraction`, `list`, etc.
Consultez l'aide : `?type;`
Les variables locales sont celles introduites dans la procédure et dont les noms seront « désaffectés » à la fin de la procédure.

Par exemple :

```
> Table:=proc(n::posint)
  local T,k;
  T:=[seq(n*k,k=1..10)];
  printf("La table de %a est %a",n,T)
  end:
```

Vous pourrez regarder la syntaxe de `printf`. L'usage des %a indique qu'on va les remplacer dans l'ordre par les variables formelles qui sont listées après les guillemets. Pour avoir des nombres flottants on utilise %f.
Par exemple, pour obtenir la table de 5 on tape :

```
> Table(5);
```

On a souvent besoin du résultat d'une procédure et la petite phrase « pour faire joli » peut être gênante.
On préférera une procédure de cette forme :

```
> Table:=proc(n::posint)
  local k;
  RETURN([seq(n*k,k=1..10)])
  end:
```

On peut ainsi afficher le deuxième élément de la table de 5 :

```
> Table(5)[2];
```

Que donne `Table(-5)` ?
On peut utiliser une structure conditionnelle :

```
> parite:=proc(n::integer)
   if (irem(n,2)=0)
     then RETURN('pair')
     else RETURN('impair')
   fi
  end:
```

À votre avis, que renvoie `irem(a,b)` ?

V - Programmation impérative

C'est le type de programmation le plus répandu. On agit directement sur la mémoire de l'ordinateur pour essayer de l'économiser à l'aide d'affectations. Nous verrons que cela peut poser certains problèmes.

a. Boucles itératives

On veut effectuer une action un certain nombre de fois ; on utilise une instruction du type :

```
for var from debut to fin by increment do
    instructions;
od;
```

Par exemple :

```
> Somme:=proc(n::posint)
  local S,k;
  S:=0;
   for k from 1 to n by 1 do
     S:=S+2*k-1;
   od;
 RETURN(S)
 end:
```

On peut arrêter l'itération dès qu'une condition a été réalisée (et se passer du to) :

```
> Produit:=proc(n::posint)
  local P,k;
  P:=1;
   for k from 1 while P<n do
     P:=P*k;
    print(k,P);
   od;
 end:
```

Expliquez....
On peut même se passer du for :

```
> quoquo:=proc(a::nonnegint,b::posint)
  local q;
  q:=1;
    while q*b<=a do
      q:=q+1;
    od;
  RETURN(q-1);
  end:
```

Expliquez...

VI - Programmation récursive

On créé des procédures qui s'appellent elles-mêmes, un peu selon le principe des suites définies par une relation $u_{n+1} = f(u_n)$ mais de manière plus générale.
Reprenons un exemple du paragraphe précédent :

```
> Somme_r:=proc(n::posint)
    if n=1 then 1
    else Somme_r(n-1)+2*n-1;
    fi;
end:
```

Il faut connaître un cas simple et un moyen de simplifier un cas compliqué...
On peut obtenir les résultats intermédiaires avec l'option remember et en demandant le quatrième opérande de la procédure :

```
> Somme_r:=proc(n::posint)
    option remember;
    if n=1 then 1
    else Somme_r(n-1)+2*n-1;
    fi;
end:
```

Puis :

```
Somme_r(9);
op(4,eval(Somme_r));
```

Cette option est malgré tout plus intéressante dans le cas de récursions multiples : les calculs intermédiaires étant stockés, ils n'ont pas à être recalculés et cela gagne du temps.
Par exemple, pour la suite de Fibonacci $u_{n+2} = u_{n+1} + u_n$ avec $u_0 = u_1 = 1$:

```
> fibo:=proc(n::nonnegint)
   option remember;
   if n=0 then RETURN(1);
     elif n=1 then RETURN(1);
     else RETURN(fibo(n-1)+fibo(n-2));
   fi
  end:
```

Sans l'option remember, fibo(30) prend pas mal de temps mais est instantané quand on l'ajoute : à ne pas oublier...

VII - Exercices

Exercice 1 Partie entière

Trouver un algorithme récursif puis un algorithme impératif déterminant la partie entière d'un entier.

- dans le cas récursif, on se souviendra que la partie entière d'un nombre appartenant à $[0;1[$ est nulle, que $[x] = 1 + [x-1]$ et que $[x] = -1 + [x+1]$;
- dans le cas impératif, on partira de zéro et on se promènera sur l'axe des réels en faisant des pas de longueur 1.

Exercice 2 Calculs de sommes

Donner des procédures impératives et récursives prenant comme argument une fonction f un entier n_0 et un entier n et qui calcule $\sum_{k=n_0}^{n} f(k)$.

Exercice 3 Maximum

Donner des procédures impératives et récursives donnant le maximum d'une liste de nombres réels quelconques.

Exercice 4 Tête et queue

La tête d'une liste est son premier élément. La queue d'une liste est la sous-liste créée en lui coupant la tête. Donnez deux procédures `tete(liste)` et `queue(liste)`.

Exercice 5 Test de croissance

Donner des procédures impératives et récursives déterminant si une liste donnée est croissante. On pourra éventuellement utiliser `tete` et `queue`.

Exercice 6 Dichotomie

Pour résoudre une équation du type $f(x) = 0$, on recherche graphiquement un intervalle $[a,b]$ où la fonction semble changer de signe.
On note ensuite m le milieu du segment $[a,b]$. On évalue le signe de $f(m)$.
Si c'est le même que celui de $f(a)$ on remplace a par m et on recommence. Sinon, c'est b qu'on remplace et on recommence jusqu'à obtenir la précision voulue.
Donner une procédure récursive puis une procédure impérative prenant comme arguments une fonction f, les bornes de l'intervalle d'étude a et b et une précision de calcul eps.

Exercice 7 Méthode des parties proportionnelles

Il peut être parfois plus efficace de ne plus considérer le milieu de $[a;b]$ mais l'abscisse de l'intersection de la droite passant par les points $(a,f(a))$ et $(b,f(b))$ avec l'axe des abscisses. Modifiez légèrement vos procédures précédentes et comparez.

Exercice 8 Algorithme de Héron

Héron d'Alexandrie a trouvé une méthode permettant de déterminer une approximation de la racine carrée d'un nombre positif vingt siècles avant l'apparition des ordinateurs.
Si x_n est une approximation strictement positive par défaut de $\sqrt{a}$, alors a/x_n est une approximation par excès de $\sqrt{a}$ (pourquoi ?) et vice-versa.
La moyenne arithmétique de ces deux approximations est $\frac{1}{2}\left(x_n + \frac{a}{x_n}\right)$ et constitue une meilleure approximation que les deux précédentes.
On peut montrer c'est une approximation par excès (en développant $(x_n - \sqrt{a})^2$ par exemple).
On obtient naturellement un algorithme... Donnez une version récursive puis impérative prenant en argument le nombre dont on cherche la racine carrée, une première approximation x_0 et une précision eps.

Exercice 9 Calcul de 300 décimales de π

Posons $e_n = 1 + \frac{1}{1!} + \frac{1}{2!} + \cdots + \frac{1}{n!}$
Alors on a aussi

$$e_n = 1 + 1 + \frac{1}{2}\left(1 + \frac{1}{3}\left(1 + \frac{1}{4}\left(\cdots\left(\frac{1}{n-1}\left(1 + \frac{1}{n}\right)\right)\right)\right)\right)$$

Un exercice classique montre que $e - e_n \leqslant \frac{n+2}{n+1}\frac{1}{(n+1)!}$.
Ainsi, pour $n = 167$, on obtiendra 300 bonnes décimales au moins : trouvez-les !

Exercice 10 Méthode des rectangles

Trouvez une version récursive et une version impérative d'un algorithme calculant une approximation d'une intégrale par la méthode des rectangles étant donné f, les bornes a et b et dx